Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ

Frederick Reiss, and Joseph M. Hellerstein

IRB-TR-04-004

February, 2004

Intel Research Berkeley

Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ

Frederick Reiss* and Joseph M. Hellerstein*†

*U.C. Berkeley Department of Electrical Engineering and Computer Science and †Intel Research, Berkeley

Abstract

Many of the data sources used in stream query processing are known to exhibit bursty behavior. Data in a burst often has different characteristics than steady-state data, and therefore may be of particular interest. In this paper, we describe the *Data Triage* architecture that we have added to TelegraphCQ to react to such bursts.

When a burst of data requires load-shedding, TelegraphCQ chooses tuples to remove from the data flow. The Data Triage component then constructs synopses of these tuples and uses a fast but approximate *shadow query plan* to estimate the query results that the system did not have time to compute. These results are then combined with the system's standard query results to capture the properties of the entire input.

We describe how we leveraged the object-relational features of TelegraphCQ to implement Data Triage entirely outside the system's standard-case query engine. Through a series of experiments using real-time measurements of the system, we show that Data Triage provides more accurate query results than competing load-shedding methodologies across a wide range of loads.

1 Introduction

One of the distinguishing properties of stream query processors is that they produce query results in real time. For applications like market analysis, network monitoring, and inventory tracking, timely query results are of great importance.

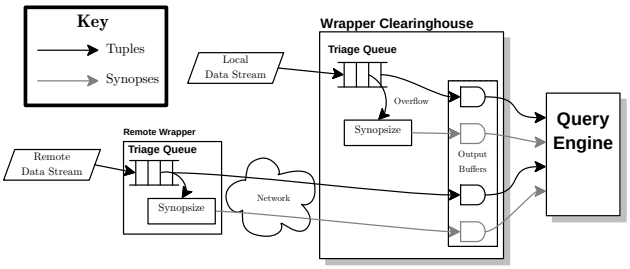

Figure 1: The Data Triage load-shedding architecture. We embed *triage queues* inside the gateway modules that convert data streams into the system's internal format. If the query engine cannot consume tuples at the rate they enter the triage queues, excess tuples are synopsized and the synopses sent to the engine.

The requirement for low result latency raises design challenges, since query processors must return useful results quickly regardless of the rate at which they receive data. Much recent work has focused on methods of coping with excessive data rates in streaming query processors by *shedding load*.

Of course, a well-configured system should have enough capacity to handle its expected steady-state load. Studies show that common sources of streaming data – network traffic, environmental monitoring, software logs, etc. – often exhibit "bursty" behavior [21] [30]. Bursty behavior is characterized by periods of low data rates punctuated by "bursts" of high data rates that vary in their length and speed. Available network bandwidth and incoming query workloads may also be affected during periods of bursts, leading to a situation in which the effective load on a stream query processor can vary rapidly and unpredictably by orders of magnitude.

We expect, therefore, that streaming query processors will use their load-shedding mechanisms mostly during transient bursts of very high system load. Note that bursts often produce not only *more* data, but also *different* data than usual. This will often be the case, for example, in crisis scenarios (network attacks,

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

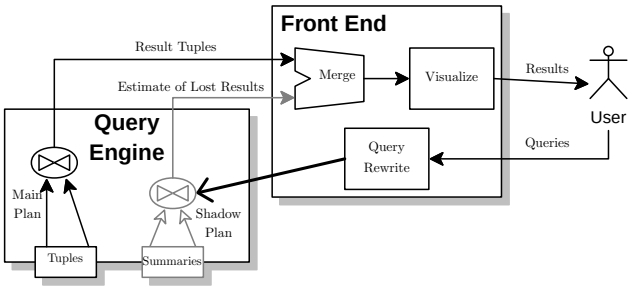

Figure 2: Quantifying the effects of dropped tuples using a shadow query plan. The *main query plan* on the left operates on relational tuples, while the *shadow plan* on the right runs with synopsis data structures.

environmental incidents, software malfunctions, etc.), where a high volume of unusual readings may be reported to the system. Hence, analysts may be particularly eager to capture the properties of the data in the burst. This type of usage poses several design challenges for these mechanisms. The load-shedding mechanisms need to react quickly to changes in load. These mechanisms need to produce accurate query results with low latency across a wide range of system loads. Load-shedding should not simply discard data; it must capture properties of the missing data. Since the system will not be shedding load most of the time, the mechanism for doing so must not interfere with the "standard-case" processing of data.

We have developed a load-shedding architecture called Data Triage that is designed for bursty loads. Data Triage addresses the challenges of these usage scenarios by:

- Responding quickly to changes in load
- Providing relatively accurate query results across a wide variety of data rates and available amounts of bandwidth
- Compensating for situations in which the data in a burst has different characteristics from the data outside the burst
- Keeping load-shedding logic outside the main query processing datapath and close to the data source in scenarios where distributed gateways can be deployed.

Figures 1 and 2 give an overview of of our architecture. Data triage places a *triage queue* between each data source and the query processor. During normal operation, data sources push tuples onto their respective queues, and the query processor pulls tuples off the back of the queues as needed. If the speed of the data streams exceeds the capacity of the query processor to process data, then the triage queues fill up. When a triage queue runs out of space, the system uses a drop policy to remove less-critical tuples from the queue, and uses synopses to capture the approximate properties of the deleted set of tuples. At the end

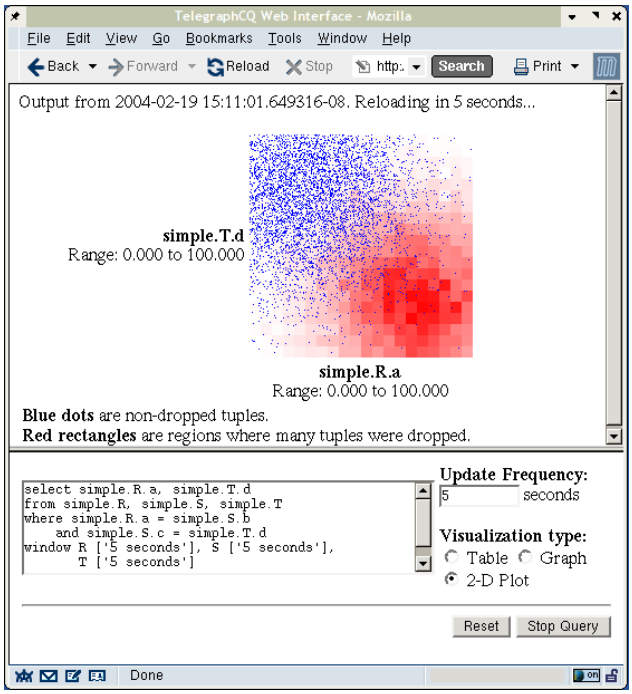

Figure 3: Data triage running with a TelegraphCQ web interface. The query in this screenshot returns two-dimensional tuples, and the user has chosen a 2-D visualization. The visualization shows query results as blue points and the system's estimate of lost result tuples as rectangles in varying shades of red. Note that the colors in this image are present in the digital version of this paper, but may not be in the printed version.

of each time window in a continuous query, the triage subsystem passes these synopses to the query engine, which uses a *shadow query plan* to compute an approximate version of the query results it did not have time to compute exactly. Finally, the exact and approximate results are merged to produce a composite query result for the window.

In this paper, we describe our implementation of Data Triage on the TelegraphCQ stream query processor. A novel feature of our design is that most of the functionality of Data Triage is implemented using query rewriting and the object-relational features of the query engine. In addition to avoiding substantial modifications to the core query engine of TelegraphCQ, this strategy provides us with a theoretical framework that ensures the correctness of our solution.

We evaluate our design via experiments on our TelegraphCQ implementation, comparing Data Triage to a tuple-dropping scheme and a synopsis-only scheme. Our results indicate that Data Triage dominates the other two schemes by a significant margin across a variety of data rates in both static and bursty settings.

The rest of this paper is structured as follows: In Section 2, we briefly summarize related work. In Sec-

tion 3, we describe the properties of the relational algebra upon which our query rewrite is based. In Section 4, we develop our query rewriting algorithm, starting from our algebra and finishing with a rewrite from SQL to SQL. Section 5 describes our implementation, first explaining how we applied an object-relational approximation to our query rewrite and then exploring the systems issues we confronted. Sections 6 and 7 describe our experiments and their results. Finally, Section 8 summarizes the accomplishments of this paper and describes future work.

2 Related Work

Much work in load shedding for stream query processing has focused on methods for dropping tuples from the dataflow [36, 7, 18]. Some papers concentrate on methods for choosing load shedding points within a dataflow network, while others focus on heuristics for choosing which tuples to drop. Researchers have proposed several methods of selectively choosing which tuples to drop, ranging from dropping arbitrary tuples to using unbiased sampling methods. As an alternative to dropping tuples, a variety of work has proposed using *synopsis data structures* to deal with high load [4, 20]. Load shedding systems that use synopses lossily compress sets of tuples and perform query processing on the compressed sets. The STREAM data manager [9] uses either dropping or synopses to handle load. Garofalakis and Gibbons give a good overview of work in this area [10]. In general, this previous work has focused on situations in which the steady-state workload of the query processor exceeds its capacity to process input streams.

Synopsis data structures themselves are well-studied in the field of database research, and we highlight some of the work here. Researchers have developed many different kinds of multidimensional histograms for database applications [32, 24, 16, 3, 8, 35]. Chakrabarti *et al.* [5] describe methods of performing query processing entirely in the wavelet domain using multidimensional wavelets. Getoor *et al.* [11] use probabilistic graph models as a compressed representation of relations. Pavlov, Smith, and Mannila [29, 28] investigate methods for lossy summarization of large, sparse binary data sets. Wang *et al.* have studied methods of performing query processing and selectivity estimation using wavelets [22, 23, 37]. Lim *et al.* use query feedback to tune a materialized set of histograms to the workload of the database.

Many researchers have explored the use of random sampling to improve response time for database queries and to perform selectivity estimation during query optimization. Olken and Rotem [25] summarize work in the area prior to 1995. CASE-DB [27, 26] is a real-time database system that meets its real-time constraints by giving approximate answers to queries. As recent examples, Chaudhuri *et al.* [6, 2] analyze the effects of random sampling of base relations on the results of joins of those relations. Acharya *et al.* present a technique called *join synopses* that involves sampling from the join of the tables in a star schema rather than from the base tables themselves [1].

Online aggregation techniques [15, 14] take increasingly large samples from one or more base relations to produce incrementally refining approximate answers. In the case of join queries, they merge samples of varying sizes from multiple relations. This strategy bears some similarity to Data Triage, but there are basic distinctions. Our work focuses on computing a single approximate result for a real-time window of a data stream, whereas online aggregation provides incrementally refining summaries of static tables. Moreover, online aggregation focused solely on sampling the data, whereas we explicitly attempt to process all the data in a stream, to varying degrees.

Our formal foundation in Section 3 has analogs in algebraic approaches to materialized view maintenance for relations [34, 12] and streams [17]. A thorough survey of materialized view maintenance work appears in [13].

3 Useful Properties of the Relational Algebra

In this section, we develop an algebraic formalism that we use in the remainder of this paper to correctly rewrite queries into a Data Triage context. The crux of this section is to capture the effects on relational algebra expressions when tuples disappear from their base relations. We use relational algebra to build a set of *differential operators*. Each differential operator computes the changes to the outputs of the corresponding relational algebra operator. Our approach here resembles past work in maintaining materialized views [13], though our setting is different.

3.1 Notation

When a relational query processor drops tuples from its input relations, the change propagates through all of the intermediate results of its queries. We call the relations that result from these changes *noisy* relations. If S is a relation, we denote the corresponding *noisy* relation by S_{noisy}.

We wish to quantify the difference between a relation S and its corresponding *noisy* relation S_{noisy}. We model this difference with two separate relations: an *added* relation S_+ and a *dropped* relation S_-. Intuitively, S_- represents the tuples that are removed from S as a result of dropping tuples from base tables. Since relational operators like negation and set difference produce additional output tuples when their inputs decrease in size, we also need S_+ to represent tuples that are added to S.

The differential operators that follow maintain the invariant that the *noisy* version of any relation can be constructed from the three relations mentioned above:

$$S_{noisy} \equiv S + S_+ - S_- \quad (1)$$

where $+$ and $-$ are the multiset union and multiset difference operators, respectively. Note that

$$S = S_{noisy} - S_+ + S_-. \quad (2)$$

3.1.1 Operator Notation

For a given unary relational operator $F(S)$, we create a corresponding differential relational operator $\widehat{F}(S_{noisy}, S_+, S_-)$, where S is an arbitrary relation. The inputs of $\widehat{F}$ are the *noisy* , *added* , and *dropped* relations described in Section 3.1. If the output of $F(S)$ is R, then $\widehat{F}$ produces as its output a similar set of relations, (R_{noisy}, R_+, R_-).

Similarly, we define the differential relational algebra operator that corresponds to the *binary* operator G as

$$(S_{noisy}, S_+, S_-)\, \widehat{G}\, (T_{noisy}, T_+, T_-) \equiv (R_{noisy}, R_+, R_-). \quad (3)$$

3.2 Some Operators

In this section, we define a set of differential operators that correspond to the relational algebra operators $\langle \sigma, \pi, \times, - \rangle$.

3.2.1 Selection

We create the differential selection operator $\widehat{\sigma}(S_{noisy}, S_+, S_-) \equiv (R_{noisy}, R_+, R_-)$ from the standard selection operator σ as follows:

$$\widehat{\sigma}(S_{noisy}, S_+, S_-) \equiv (\sigma(S_{noisy}), \sigma(S_+), \sigma(S_-)). \quad (4)$$

Intuitively, the differential selection operator simply applies traditional selection to all three channels.

3.2.2 Projection

The definition of the differential projection operator is similar to that of the differential selection operator:

$$\widehat{\pi}(S_{noisy}, S_+, S_-) \equiv (\pi(S_{noisy}), \pi(S_+), \pi(S_-)). \quad (5)$$

It is important to note that the differential projection operator only works properly for multisets. We revisit the problem of handling `SELECT DISTINCT` queries in our Future Work section.

3.2.3 Cross Product

The definition of the differential cross-product operator is more complicated than the previous two operators.

Since

$$S_{noisy} \equiv S + S_+ - S_- \quad (6)$$
$$T_{noisy} \equiv T + T_+ - T_-, \quad (7)$$

for any relations S and T, we have

$$\begin{aligned} S \times T &= S_{noisy} \times T_{noisy} \\ &\quad -S_+ \times T_+ \\ &\quad -S_+ \times (T_{noisy} - T_+) \\ &\quad -(S_{noisy} - S_+) \times S_+ \\ &\quad +S_- \times T_- \\ &\quad +S_- \times (T_{noisy} - T_+) \\ &\quad +(S_{noisy} - S_+) \times T_-). \end{aligned}$$

Accordingly, we define the differential cross product operator $\widehat{\times}$ as

$$(S_{noisy}, S_+, S_-)\widehat{\times}(T_{noisy}, T_+, T_-) \equiv (R_{noisy}, R_+, R_-) \quad (8)$$

where

$$R_{noisy} = S_{noisy} \times T_{noisy}$$

and

$$\begin{aligned} R_+ &= S_+ \times T_+ \\ &\quad +S_+ \times (T_{noisy} - T_+) \\ &\quad +(S_{noisy} - S_+) \times T_+ \end{aligned}$$

and

$$\begin{aligned} R_- &= S_- \times T_- \\ &\quad +S_- \times (T_{noisy} - T_+) \\ &\quad +(S_{noisy} - S_+) \times T_-). \end{aligned}$$

3.2.4 Join

The derivation of the differential join operator follows the same line of reasoning as that of the differential cross product operator and produces essentially the same definition. For brevity, we omit this derivation.

3.2.5 Set Difference

The set difference operator has the interesting property that removing tuples from one of its inputs can *add* tuples to its output. We model this behavior by defining the differential set difference operator $\widehat{-}$ as

$$(S_{noisy}, S_+, S_-)\widehat{-}(T_{noisy}, T_+, T_-) \equiv (R_{noisy}, R_+, R_-), \quad (9)$$

where

$$R_{noisy} = S_{noisy} - T_{noisy}$$

and

$$\begin{aligned} R_+ &= (S_+ - T_{noisy}) \\ &\quad + ((T_- - S_+) \cap S_{noisy}) \end{aligned}$$

and

$$\begin{aligned} R_- &= (S_+ \cap T_-) \\ &\quad + ((S_{noisy} \cap T_+) - S_+) \\ &\quad +(S_- - T_- - T_{noisy}) \end{aligned}$$

4 Query Rewrite

An important goal in adding a load-shedding architecture to our system was to make as few changes as possible to our query optimizer and query processing engine. To meet this goal, we devised a method of rewriting SQL queries to compute the *added* and *dropped* portions of the query result.

In this section, we describe this query rewrite in detail. In Section 4.1, we start with a two-stage rewrite from relational algebra to relational algebra. In Section 4.2 apply this general rewrite to a generic select-project-join (SPJ) query to obtain an SQL query rewrite.

4.1 Relational Algebra Rewrite

In general, one can rewrite any relational algebra query to compute the corresponding *dropped* and *added* relations by applying the following transformations:

1. Replace each operator F in the expression with its equivalent differential version $\widehat{F}$ to create a *differential query*.
2. Convert the differential query back into relational algebra by recursively applying the operator definitions in Section 3.2. This step produces three different queries.

For a query whose relational algebra representation consists only of join operators, a more straightforward rewriting exists. We derive this rewriting in the following section.

4.2 Select-Project-Join Queries

In this section, we use the method of Section 4.1 to generate a query that returns the *dropped* results of a general select-project-join (SPJ) query [1]. For ease of exposition, we omit selection and projection operators from our equations, concentrating on the more difficult join operator.

Consider an arbitrary relational algebra query Q consisting of the join of n relations:

$$Q \equiv R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n. \tag{10}$$

We will now convert Q into the corresponding *dropped* query using the query rewrite in Section 4. The the first stage of the rewrite produces a differential relational algebra query for Q:

$$R_{1noisy} \widehat{\bowtie} R_{2noisy} \widehat{\bowtie} \ldots \widehat{\bowtie} R_{nnoisy}. \tag{11}$$

The second stage of the rewrite uses the definitions in Section 3.2 to convert the *dropped* query back into standard relational algebra. This conversion produces three queries, one for each of Q_{noisy}, Q_- and Q_+.

[1]Since SPJ queries contain no set difference or negation operators, they have no *added* results when tuples are dropped from their inputs.

The the first query, which computes Q_{noisy}, is straightforward:

$$Q_{noisy} = R_{1noisy} \widehat{\bowtie} R_{2noisy} \widehat{\bowtie} \ldots \widehat{\bowtie} R_{nnoisy}$$

The second query, which computes Q_-, is significantly more difficult to derive. We start by deriving a recurrence relation for this query, and then expand the recurrence to compute the query.

For any $1 \leq i, j \leq n$, $i \leq j$, let $\mathcal{R}_{i,j}$ denote the join of relations R_i through R_j, inclusive. The following recurrence relation for $\mathcal{R}_{1,k\_}$ follows from a straightforward application of the definition in Section 3.2.3:

$$\begin{aligned} \mathcal{R}_{1,k\_} &= R_{1\_} \bowtie \mathcal{R}_{2,k} + ((R_{1noisy} - R_{1+}) \\ &\quad \bowtie \mathcal{R}_{2,k\_}). \end{aligned}$$

Expanding by this recurrence gives us:

$$\begin{aligned} Q_- &= \mathcal{R}_{1,n\_} \\ &= R_{1\_} \bowtie \mathcal{R}_{2,n} + ((R_{1noisy} - R_{1+}) \\ &\quad \bowtie \mathcal{R}_{2,n\_}) \\ &= R_{1\_} \bowtie \mathcal{R}_{2,n} + ((R_{1noisy} - R_{1+}) \\ &\quad \bowtie (R_{2\_} \bowtie \mathcal{R}_{3,n} + ((R_{2noisy} - R_{2+}) \\ &\quad \bowtie \mathcal{R}_{3,n\_}))) \\ &= \ldots \\ &= R_{1\_} \bowtie \mathcal{R}_{2,n} + ((R_{1noisy} - R_{1+}) \\ &\quad \bowtie (R_{2\_} \bowtie \mathcal{R}_{3,n} + ((R_{2noisy} - R_{2+}) \\ &\quad \bowtie (R_{3\_} \bowtie \mathcal{R}_{4,n} + ((R_{3noisy} - R_{3+}) \\ &\quad \bowtie (\ldots + (R_{n-1noisy} - R_{n-1+}) \bowtie R_{n\_})) \ldots). \end{aligned}$$

Similarly, we can derive an expression for the *added* relation Q_+ using the recurrence:

$$\begin{aligned} \mathcal{R}_{1,k+} &= R_{1+} \bowtie \mathcal{R}_{2,k_{noisy}} + ((R_{1noisy} - R_{1+}) \\ &\quad \bowtie \mathcal{R}_{2,k+}), \end{aligned}$$

Note that the values of the expressions for Q_- and Q_+ can both be computed with $3n-1$ join operations by reusing intermediate results.

If no new tuples are added to the relations in the query, then $R_{i+} = \emptyset$ for all $1 \leq i \leq n$. Thus, we have:

$$Q_{noisy} = R_{1noisy} \widehat{\bowtie} R_{2noisy} \widehat{\bowtie} \ldots \widehat{\bowtie} R_{nnoisy} \tag{12}$$

$$\begin{aligned} Q_- = {} & R_{1\_} \bowtie \mathcal{R}_{2,n} + (R_{1noisy} \\ & \bowtie (R_{2\_} \bowtie \mathcal{R}_{3,n} + (R_{2noisy} \\ & \bowtie (R_{3\_} \bowtie \mathcal{R}_{4,n} + (R_{3noisy} \\ & \bowtie (\ldots + R_{n-1noisy} \bowtie R_{n\_})) \ldots) \end{aligned} \tag{13}$$

$$Q_+ = \emptyset. \tag{14}$$

4.3 An Example

In this section, we demonstrate the query rewriting process using a simple example query. The following SQL query computes the equijoin of three streams, `R`, `S`, and `T`:

```
SELECT *
FROM R, S, T
WHERE R.a = S.b AND S.c = T.d;
```

In this section, we will rewrite this query using the technique outlined in section 4.2. In particular, we will focus on the rewrite that deals only with dropped tuples.

We start by dividing each of the streams into two "substreams", one to represent the tuples that are dropped and one to represent the tuples that are not dropped:

```
CREATE STREAM R_dropped (a INTEGER);

CREATE STREAM R_kept (a INTEGER);

CREATE STREAM S_dropped ...
```

For convenience, we define the original streams as multiset unions of these substreams:

```
CREATE VIEW R_all AS
    (SELECT * FROM R_kept)
    UNION ALL
    (SELECT * FROM R_dropped);

CREATE VIEW S_all AS ...
```

Next, we choose a join order:

$$\texttt{Q} = \texttt{R} \bowtie \texttt{S} \bowtie \texttt{T}. \tag{15}$$

where Q represents the result of the query. We will revisit the subject of join orders in Section 5.2.

Applying the rewrite rules from equations 12 and 13 gives us the following expressions for the *noisy* and *dropped* portions of the query:

$$\texttt{Q\_kept} = \texttt{R\_kept} \bowtie \texttt{S\_kept} \bowtie \texttt{T\_kept} \tag{16}$$

$$\begin{aligned}\texttt{Q\_dropped} = {} & \texttt{R\_dropped} \bowtie \texttt{S} \bowtie \texttt{T} \\ & + (\texttt{R\_kept} \bowtie (\texttt{S\_dropped} \bowtie \texttt{T} \\ & \quad + \texttt{S\_kept} \bowtie \texttt{T\_dropped}))\end{aligned} \tag{17}$$

Recall that Q_dropped represents the tuples that disappear from the result of the query when the tuples in R_dropped, S_dropped, and T_dropped are removed from R, S, and T, respectively. Q_kept denotes the tuples that remain in the query result.

We can convert these relational algebra expressions into SQL views as shown in Figure 4.

Computing Q_dropped efficiently over relational tables would require an additional phase of query rewrite using a sophisticated engine like that in [31]. Fortunately, we do not need such a facility for our purposes, as we will show in the next section.

```
CREATE VIEW Q_kept AS
SELECT *
FROM R_kept, S_kept, T_kept
WHERE R_kept.a = S_kept.b AND S_kept.c = T_kept.d;

CREATE VIEW Q_dropped AS
(SELECT * FROM R_dropped, S_all, T_all WHERE a=b and c=d)
UNION ALL
(SELECT * FROM R_kept,
    ((SELECT * FROM S_dropped, T_all WHERE c=d) UNION ALL
     (SELECT * FROM S_kept, T_dropped WHERE c=d))
  WHERE a=b);
```

Figure 4: A rewritten query translated back to SQL.

5 Implementation

The query rewrite strategy outlined in the Section 4.2 produces a query that computes the same answer as the original query, but takes slightly longer to execute than the original query. Clearly, this rewrite is of little practical use by itself. Combined with synopsis-based query processing and a continuous-query system, however, the rewrite provides a powerful tool for efficiently estimating the effects of dropped tuples. This tool is an integral part of our Data Triage implementation.

In this section, we explain how we used the object-relational facilties of TelegraphCQ to implement a query rewrite that incorporates approximation techniques. We then give an in-depth description of the current state of our Data Triage implementation in TelegraphCQ.

5.1 Object-Relational Approximation

In this section, we describe a second stage of query rewriting that converts the results of Section 4.2 into an efficient approximate query. For ease of exposition, we will focus on the example query from Section 4.3, evaluated over a moving time window. The extension to arbitrary select-project-join queries is straightforward. We use PostgreSQL syntax [33] for declarations.

To make our query rewrite possible, we first use the object-relational facilities of our query processor to define datatypes for our synopsis data structures. In this section, we will refer to this datatype by the abstract name "Synopsis"; later in the paper we will describe experimental results using simple multidimensional histograms as our synopsis implementation.

We also create user-defined functions to perform various kinds of relational algebra operations on these synopsis data structures, such as:

```
-- Project S down to the indicated columns.
CREATE FUNCTION project(S Synopsis,
                       colnames CSTRING)
        RETURNS Synopsis
        AS ...

-- Approximate SQL's UNION ALL construct.
CREATE FUNCTION union_all(S Synopsis,
                          T Synopsis)
        RETURNS Synopsis
```

```
        AS ...

-- Compute approximate equijoin of S and T.
CREATE FUNCTION equijoin(S Synopsis,
                         S_colname CSTRING,
                         T Synopsis,
                         T_colname CSTRING)
        RETURNS Synopsis
        AS ...
```

We rewrite `CREATE STREAM` statements to create auxiliary streams of synopses:

```
-- Original statement was:
--      CREATE STREAM R(a integer);

CREATE STREAM R(a integer);

-- Synopsis of dropped tuples.  <earliest>
-- and <latest> indicate the timestamp range
-- of the tuples synopsized.
CREATE STREAM R_dropped_syn(syn Synopsis,
                            earliest Timestamp,
                             latest Timestamp);
```

To avoid writing functions for joining synopses with normal tuples, we chose to have TelegraphCQ generate synopses of the tuples it processes, so that we could use these synopses in approximating the lost query results. We create an additional stream to hold these synopses of "kept" tuples:

```
CREATE STREAM R_kept_syn ...
```

The overhead of creating these additional synopses can be quite small – in our experimental framework, the cost of forming and manipulating synopses is dwarfed by the cost of standard-case query processing. (See Figure 6). Of course some synopses are more complex to create and manipulate than others; we return to this point in Section 8.1.

Finally, we apply the formulas in Equations 16 and 17 to obtain the views in Figure 5.

In spite of its size, this expression can be evaluated quite quickly due to the efficiency of synopsis-based query processing. To measure the overhead of computing this expression, we developed a microbenchmark to compare the performance of the sample query in this section with the rewritten query above. We replaced the references to `R_d.syn` and the like with calls to user-defined functions for building multidimensional histograms from tables. We used two kinds of synopsis data structure in this microbenchmark: An untuned implementation of the MHIST data structure [32], and a more efficient but simpler sparse multidimensional histogram implementation. We ran the queries on tables loaded with 10000 randomly-generated tuples each and compared their performance. Figure 6 gives the results of this comparison. We found that, provided that a sufficiently fast synopsis data structure is used, the rewritten query runs in a small fraction of the time of the original query. We revisit synopsis data structure efficiency in the Experiments section of this paper.

```
CREATE VIEW Q_kept AS
SELECT * FROM R_kept, S_kept, T_kept
WHERE R_kept.a = S_kept.b AND S_kept.c = T_kept.d;

CREATE VIEW Q_dropped AS
SELECT
union(
  equijoin(R_d.syn, 'R.a',
    equijoin(union(S_d.syn, S_k.syn), 'S.c',
             union(T_d.syn, T_k.syn), 'T.d'), 'S.b'
  ),
  equijoin( R_k.syn, 'R.a',
    union(equijoin( S_d.syn, 'S.c', union(T_d.syn, T_k.syn), 'T.d'),
          equijoin(S_k.syn, 'S.c', T_d.syn, 'T.d')
    ), 'S.b'
  )
)
as result
FROM R_kept R_k, R_dropped R_d, S_kept S_k, S_dropped S_d,
       T_kept T_k, T_dropped T_d
WINDOW R_k ['1 second'], R_d ['1 second'], S_k ['1 second'],
     S_d ['1 second], T_k ['1 second'], T_d ['1 second'];
```

Figure 5: Final result of rewriting the sample query to calculate an approximate version of what tuples were lost from its results. Since each synopsis stream generates a single tuple per window, the cross-product in this query only produces one tuple per window.

5.2 Join Orders for the *Dropped* Channel

When we derived our SQL query rewrite in Section 4.3, we chose a join order for the streams `R`, `S`, and `T` without explaining our choice in detail. The join ordering problem is quite different when one is performing query processing over synopsis data structures instead of over relations. One reason for this difference is that the size of the synopsis of a relation depends not on the number of tuples in the relation but on the structure of the synopsis: for example, the number and positions of bins in a histogram, or the choice of a wavelet basis function and the coefficients retained after truncation. Previous work [8] has identified an algorithm for choosing efficient join orders over dense multidimensional histograms; we return to this point in Section 8.1.

5.2.1 Current Status

In this section, we describe the current implementation status of Data Triage in the TelegraphCQ system. The implementation described in this section is currently part of the internal development build of TelegraphCQ. We plan to include Data Triage as part of a public TelegraphCQ release in Fall 2004.

TelegraphCQ supports three methods of shedding load:

- A *drop-only* method that drops tuples in response to overload conditions.
- A *summarize-only* method that builds synopses of all input data and performs approximate query processing over those synopses.

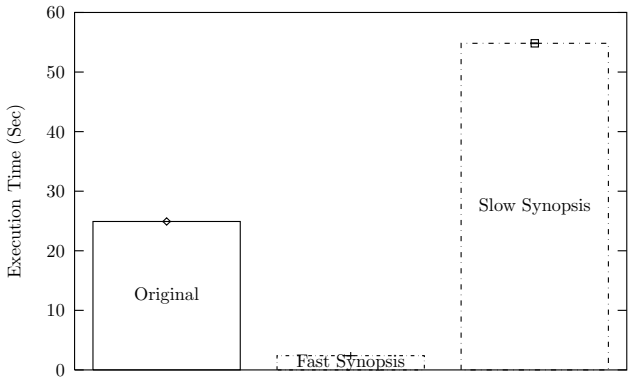

Figure 6: Results of comparing the overhead of a synopsized version of our example 3-way join query with the original query. The results demonstrate that, with an efficient synopsis data structure implementation, the Data Triage approch of synopsizing the *dropped* tuples adds minimal overhead to query processing.

- The *data triage* method described in this paper.

To implement the three methods with as much shared code as possible, we first implemented the Data Triage architecture depicted in Figure 1 on top of TelegraphCQ. Data Triage centers around a priority queue that drops tuples in response to spikes in load and generates summaries of dropped data. To implement drop-only load shedding, we disabled the code that computes summaries. To implement summarize-only load shedding, we bypassed the queue and constructed summaries of all the tuples in each stream.

Our single codebase enables a reasonably fair comparison of the three load-shedding methodologies. In particular, our Data Triage implementation uses the same tuple-dropping mechanism as our drop-only load-shedding implementation, and the same synopsis mechanism as our summarize-only implementation.

We implemented our synopsis data structures as user-defined SQL datatypes, and we wrote all operations on synopsis data structures as user-defined functions. As a result, we are able to express all computations on these synopsis data structures as SQL queries that ran inside the TelegraphCQ query engine.

Our current implementation places the logic for merging query results with summaries inside a TelegraphCQ web interface (See Figure 3). The web interface engine also invokes our query rewriter in the current implementation. To support custom applications better, we plan to integrate the rewrite and merge functionality into the query engine in a future version of TelegraphCQ.

The current build of TelegraphCQ uses a random drop policy. When our triage queue reaches its capacity, it chose a victim at random from the tuples in its buffer. We discuss alternatives to this random policy in Section 8.1.

5.2.2 Synopsis Data Structures

Synopsis data structures for stream query processing need to meet stringent performance requirements in order to be useful:

- The cost of inserting a tuple into the data structure must be significantly less than the incremental cost of performing normal join processing on the tuple.
- The data structure must perform join operations quickly and produce compact synopses of join results.

For the experimental results presented in this paper, we used a sparse multidimensional histogram with cubic buckets. We implemented this data structure within TelegraphCQ's object-relational framework, as described in Section 5.1.

We also implemented an *MHIST* multidimensional histogram using the MAXDIFF heuristic to perform bucket splits [32]. Our implementation gave more accurate query results at a given data structure size, but its performance on join queries was not sufficiently fast to keep up with the data rates in our experiment.

Profiling quickly revealed the problem: When the bucket boundaries of MHISTs are not aligned, computing their join can produce a quadratic number of new buckets. By modifying MHISTs with one of a variety of alignment constraints, we could work around this problem. However, we will see that our simple histograms work reasonably well for the purposes of Data Triage; equally-efficient but higher-accuracy histograms would improve our results for both data triage and summarize-only schemes.

6 Experiments

In this section, we present a set of experiments aimed at verifying the predicted benefits of Data Triage compared with load-shedding mechanisms based solely on dropping or summarizing tuples.

Section 6.1 gives an intuitive summary of our experimental hypothesis. Sections 5.2.1 through 6.2.2 describe our experimental setup and give more specific definitions of the terms in our hypothesis. Section 6.3 gives a description of the method we used to compare the quality of query results. Section 7 describes the results of our experiments.

6.1 Experimental Hypothesis

Our experiments compared the Data Triage approach to load-shedding outlined in this paper with alternatives based on dropping excess data or summarizing all data. We designed the experiments to test whether Data Triage meets its design goals with regard to the accuracy of query results. Intuitively, these goals are:

- Under constant low load, Data Triage should produce query results as accurate as those of drop-only load-shedding.

```
SELECT a, COUNT(*) as count
FROM R,S,T
WHERE R.a = S.b AND S.c = T.d
GROUP BY a;
WINDOW R['1 second'], S['1 second'], T['1 second'];
```

Figure 7: TelegraphCQ SQL text of the query used in our experiments. To keep the number of results per window constant, we varied the window sizes with the speed of the data streams.

- Under constant high load, Data Triage should produce query results as accurate as those of summarize-only load-shedding.
- Under a bursty load with data in the bursts coming from a different distribution from data outside the bursts, Data Triage should provide more accurate query results than either drop-only or summarize-only load-shedding.

The sections that follow describe the specific implementations of the three load-shedding strategies that we used in our experiments, as well as the specific query and data workloads used and the method we used to compare query result quality.

6.2 Experimental Setup

Our experiments adimed at a comparative evaluation of three different load-shedding strategies. To ensure a fair comparison, we implemented the three strategies on top of the same query engine, with almost all code shared between the implementations (See Section 5.2.1). Our production implementation placed the logic for merging query results with estimates of lost results inside the web interface. For our experiments, we wrote a special driver program that ran this code outside of the web interface engine.

We ran our experiments on a server running Linux 2.6.3 on a 1.4 GHz Pentium 3 processor and 1.5 GB of main memory. While the experiments were running, the computer was in multiuser mode but with only one user logged in.

6.2.1 Query

We compared the performance of the different load-shedding techniques on a variant of the query used in Section 4.3. The text of the query in TelegraphCQ's dialect of SQL is given in Figure 7. We ran the query through our query rewriting software, producing a *dropped* query similar to the one in Figure 5.

We generated equal numbers of random tuples for each of the streams `R`, `S`, and `T` from Gaussian distributions. The fields in the tuples took on values ranging from 1 to 100, inclusive.

6.2.2 Data Rates

To simulate varying data bandwidth, we wrote a program that read raw tuples off of disk and sent them to TelegraphCQ with arbitrary time delays between tuple deliveries. We used this program to simulate both steady and bursty data arrival rates. When we simulated bursty arrival rates, the "burst" tuples were drawn from Gaussian distributions with means at different locations. This use of independent distributions was intended to test the third part of our experimental hypothesis (See Section 6.1.). Each run of the experiment used a different random seed to generate tuples.

During each run of our experiments, we varied the rate of data arrival by a series of constant factors. To keep results comparable across different data arrival rates, we scaled the size of our time windows with data arrival rate. Consequently, the number of *tuples per window* was held constant across experiments, even as we changed the data rates.

We used a simple two-state Markov model to determine which tuples were "burst" tuples and which were "non-burst" tuples. Overall, 60 percent of stream tuples were from a burst, and the expected burst length was 200 tuples. Data in bursts arrived 100 times as quickly as non-burst data. Each run of the experiment used a different random seed for determining which tuples were "burst" tuples and which were "non-burst" tuples.

6.3 Measuring Result Quality

Measuring the quality of approximate query results is an inexact science [7]. We used the following method to compare the accuracy of query results for the purposes of our experiments: We first computed the result of the query from the original data. This "ideal" result consisted of a set of aggregate values grouped by window number and various other attributes. For each group in our actual query results, we compared the aggregate value with the corresponding value from the "ideal" query result. We then computed the root mean square (RMS) value of this difference over all the groups. To avoid perturbing the timing of our experiments, we computed the RMS error offline using dumps of query results.

The reader should note that RMS error is not a linear measure. For example, an RMS error of $k \times x$ is *not* k times as bad as an RMS error of x in any clear sense. In this paper, we avoid making qualitative comparisons between RMS error levels. Instead, we restrict ourselves to statements about whether the mean error of one load-shedding method is greather than that of another by a statistically significant amount.

7 Experimental Results

In this section, we present the results of the experiments described in Section 6.2. Section 7.1 gives the results on data streams with constant data rates, and Section 7.2 gives the results on bursty data streams. These results satisfy the experimental hypothesis given in Section 6.1.

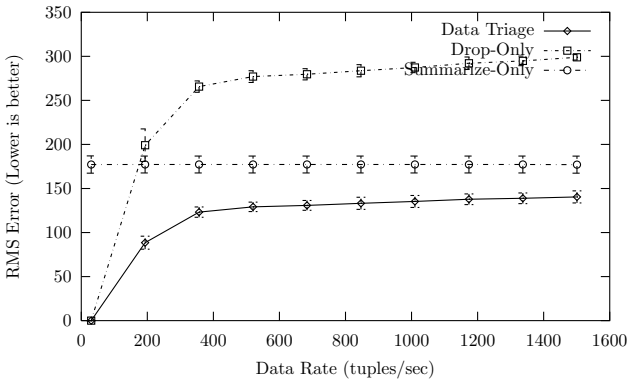

Figure 8: RMS error of query results with constant data rates. Data Triage dominates the other two load-shedding methods across a wide range of data rates. Points represent the mean of nine runs of the experiment; error bars indicate the standard deviation.

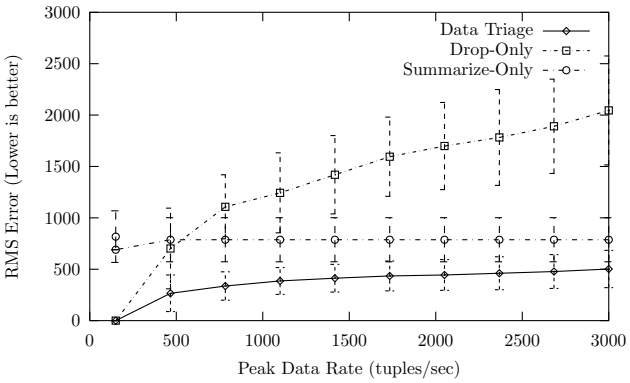

Figure 9: RMS error of query results with bursty data rates. Data Triage dominates the other two load-shedding methods by a statistically significant margin. Points represent the mean of a minimum of nine runs of the experiment; error bars indicate the standard deviation.

7.1 Constant Data Rate

Our first experiment involved sending data at TelegraphCQ at a constant rate. We gradually increased the data rate to the maximum point: when drop-only load-shedding dropped all but one triage queue's worth of tuples.

The graph in Figure 8 shows the results of this experiment. The x-axis of this graph measures the rate at which tuples entered the system. The y-axis measures the RMS error of the query results that each load-shedding method produces.

The nearly-horizontal line with circular points represents the results of the summarize-only method. As the graph demonstrates, summarizing the input data produced the same result quality across a variety of different data rates. This pattern arises because the synopses do not change with an increase in data rate; our synopsis data structure was tuned to handle the highest observed data rate.

The line with square points shows the results of using the drop-only load-shedding method. At low data rates, drop-only load-shedding is able to process every tuple that enters the system and therefore computes an entirely accurate query result. As data comes in more quickly, the drop-only method is forced to drop tuples, perturbing the query results. Eventually, the results become worse than those of the summarize-only method.

Data Triage combines the advantages of both of the other methods. At low data rates, Data Triage provides exact query results like drop-only load-shedding. As the data rate increases, Data Triage gradually falls back onto synopsis-based query processing, approaching but not exceeding the RMS error of the summarize-only method.

7.2 Bursty Data Rate

Our second experiment differed from the first in that the data rates for all the streams were bursty, and the data in the bursts came from a different random distribution from the non-burst data. As before, we gradually increased the data rate until drop-only load-shedding dropped almost all of its input tuples.

Figure 9 shows the results of this experiment. As before, the summarize-only technique produced the same results at all data rates, and the RMS error of Data Triage matches that of the drop-only technique at low data rates and that of the summarize-only technique at high data rates.

The results of the second experiment showed considerably more variance than those of the first. This increase in variance is due to the fact that the timing of tuple delivery changed from one run to the next, as each run used a different random seed.

Although the "non-burst" tuples in the second experiment were drawn from the same distribution as the tuples in the first experiment, the "burst" tuples drawn from a different distribution. Since RMS error is not a linear measure and we used different data distributions, it is difficult to make meaningful comparisons between the errors in the two runs of the experiment.

8 Conclusion

In this paper, we introduced the load-shedding strategy of Data Triage, which, by summarizing excess data instead of dropping it, improves the quality of query results under bursty data rates. We described a formally-based, practical query rewrite scheme that allows us to implement Data Triage without modifying the core of the query engine. We put forth two hypotheses which we validated experimentally via an implementation in TelegraphCQ. First, Data Triage

combines the advantages of drop-only load-shedding under low load with the advantages of synopsis-based query processing under high load. Second, Data Triage can outperform both methods under certain types of bursty load. By leveraging the object-relational extensibility that TelegraphCQ inherited from its Postgres roots, we were able to add Data Triage to TelegraphCQ with very minimal changes to the core engine.

8.1 Future Work

We are currently in the process of extending the work in this paper in several directions. One important extension of our work is to test the performance of Data Triage with additional types of synopsis data structures. We anticipate that using a more advanced synopsis into the Data Triage architecture will improve Data Triage's result quality under heavy load, as long as we take care to keep the synopsis cheap to construct and manipulate in a query. One option here is a constrained variant of MHists that picks bucket boundaries from a small finite set of options.

An additional piece of ongoing work is the implementation of new methods for choosing which tuples to drop. The design of Data Triage opens up several new possibilities for victim-selection policies. Since Data Triage synopsizes dropped tuples, it can take skewed samples of data streams without unduly skewing query results. Additionally, the triage queue can take advantage of the information in its synopses to make informed decisions about which tuples to drop. We are working to develop "synergistic" policies for our architecture, in which the triage queue chooses to drop the tuples that the synopsis data structure can summarize most efficiently.

Another issue we would like to explore further is that of merging the *kept*and *dropped* data streams that Data Triage produces. In our experiments, we merged these streams by merging the aggregates computed from a SQL `GROUP BY` statement with approximate aggregates computed from synopses. We would like to develop additional strategies for queries without aggregates. One particularly interesting possibility is that of developing special-purpose user interfaces to visualize tuples and synopses simultaneously. This is an instance of the "detail-in-context" data visualization problem (e.g., [19]). The visualization in Figure 3 represents a preliminary attempt at such an interface.

We would also like to explore the problem of processing select-project-join queries efficiently with synopsis data structures. Although there are many papers dealing with computing aggregates from synopsis data structures, relational query processing over synopses has received less attention. Deshpande and Hellerstein [8] present some preliminary results on relational processing and optimization with multidimensional histograms, but the problem merits more in-depth study.

An ambitious aspect of TelegraphCQ is its support for sharing processing across multiple continuous queries. While TelegraphCQ can naturally share processing for our *kept*tuples, we have not explored the possibility of sharing synopses of the *dropped* tuples across queries. With inexpensive synopsis schemes this may be unnecessary, but with more complex synopses this may become an important optimization.

Finally, we would like to extend our query rewriting technique to handle `SELECT DISTINCT` queries. We believe that we can perform these queries by deferring projection to the top of the shadow query plan, but we have not yet investigated the issue in depth.

Acknowledgments

References

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD Proceedings*, pages 275–286, 1999.

[2] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *SIGMOD Proceedings*, 2003.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: a multidimensional workload-aware histogram. *SIGMOD Record*, 30(2):211–222, 2001.

[4] A. Bulut and A. K. Singh. Swat: Hierarchical stream summarization in large networks. In *ICDE*, 2003.

[5] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *VLDB Journal: Very Large Data Bases*, 10(2–3):199–223, 2001.

[6] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD Proceedings*, 1999.

[7] A. Das, J. E. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD Proceedings*. ACM Press, 2003.

[8] A. Deshpande and J. Hellerstein. On using correlation-based synopses during query optimization. Technical Report CSD-02-1173, U.C. Berkeley EECS Department, May 2002.

[9] R. M. *et al.* Query processing, resource management, and approximation in a data stream management system. In *CIDR Proceedings*. ACM Press, 2003.

[10] M. Garofalakis and P. Gibbons. Approximate query processing: Taming the terabytes!, 2001.

[11] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *SIGMOD Conference*, 2001.

[12] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 328–339. ACM Press, 1995.

[13] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[14] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD Proceedings*, pages 287–298. ACM Press, 1999.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD Proceedings*, pages 171–182, 1997.

[16] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 275–286, 24–27 1998.

[17] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 113–124. ACM Press, 1995.

[18] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

[19] T. A. Keahey. The generalized detail-in-context problem. In *Proceedings IEEE Symposium on Information Visualization*, 1998.

[20] I. Lazaridis and S. Mehrotra. Capturing sensor-generated time series with quality guarantees. In *ICDE*, 2003.

[21] W. E. Leland, M. S. Taqq, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic. In D. P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[22] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD Proceedings*, pages 448–459, 1998.

[23] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *VLDB Proceedings*, 2000.

[24] M. Muralikrishna and D. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *SIGMOD Proceedings*, pages 28–36, 1998.

[25] F. Olken and D. Rotem. Random sampling from databases - a survey. In *Statistics and Computing (invited paper)*, pages 25–42, 1995.

[26] G. Ozsoyoglu, S. Guruswamy, K. Du, and W.-C. Hou. Time-constrained query processing in CASE-DB. *Knowledge and Data Engineering*, 7(6):865–884, 1995.

[27] G. Ozsoyoglu, Z. Ozsoyoglu, and W.-C. Hou. *Readings in Real-Time Systems*. IEEE Computer Society Press, 1993.

[28] D. Pavlov and H. Mannila. Beyond independence: Probilistic models for query approximation on binary transaction data.

[29] D. Pavlov and P. Smyth. Probabilistic query models for transaction data. In *Knowledge Discovery and Data Mining*, pages 164–173, 2001.

[30] V. Paxson and S. Floyd. Wide-area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226 – 224, 1995.

[31] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In M. Stonebraker, editor, *SIGMOD Proceedings*, pages 39–48. ACM Press, 1992.

[32] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *CIDR Proceedings*. ACM Press, 2003.

[33] P. Project. *Postgresql 7.4 User Manual: Server Programming*. PostgreSQL Project, 2003. `http://www.postgresql.org/docs/`.

[34] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.

[35] Y. Tao, J. Sun, and D. Papadias. Selectivity estimation for predictive spatio-temporal queries. In *ICDE*, 2003.

[36] N. Tatbul, U. Centintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB Proceedings*, 2003.

[37] M. Wang, J. S. Vitter, L. Lim, and S. Padmanabhan. Wavelet-based cost estimation for spatial queries. In *Proceedings of the SSTD conference*, 2001.